Machine Translation
02: Neural Network Basics

Rico Sennrich

University of Edinburgh
The biggest revolution in the technological landscape for fifty years
Now accepting applications!
Find out more and apply at: pervasiveparallelism.inf.ed.ac.uk

• 4-year programme: MSc by Research + PhD

• Research-focused: Work on your thesis topic from the start

• Collaboration between:
  ▶ University of Edinburgh’s School of Informatics
    ★ Ranked top in the UK by 2014 REF
  ▶ Edinburgh Parallel Computing Centre
    ★ UK’s largest supercomputing centre

• Research topics in software, hardware, theory and application of:
  ► Parallelism
  ► Concurrency
  ► Distribution

• Full funding available

• Industrial engagement programme includes internships at leading companies

Now accepting applications!
Find out more and apply at: pervasiveparallelism.inf.ed.ac.uk
Today’s Lecture

- linear regression
- stochastic gradient descent (SGD)
- backpropagation
- a simple neural network
Linear Regression

Parameters: \( \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} \)  
Model: \( h_\theta(x) = \theta_0 + \theta_1 x \)
Linear Regression

Parameters: \( \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} \)  
Model: \( h_\theta(x) = \theta_0 + \theta_1 x \)
Linear Regression

Parameters: $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$  
Model: $h_\theta(x) = \theta_0 + \theta_1 x$

$y = -5.00 + 1.50x$

Data

Population

Profit
Linear Regression

Parameters: \( \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} \)

Model: \( h_\theta(x) = \theta_0 + \theta_1 x \)

Data:

<table>
<thead>
<tr>
<th>Population</th>
<th>Profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5.00</td>
</tr>
<tr>
<td>10</td>
<td>10.00</td>
</tr>
<tr>
<td>15</td>
<td>15.00</td>
</tr>
<tr>
<td>20</td>
<td>20.00</td>
</tr>
</tbody>
</table>

\( y = -6.00 + 2.00x \)
Linear Regression

Parameters: \( \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} \)

Model: \( h_\theta(x) = \theta_0 + \theta_1 x \)
Linear Regression

Parameters: \( \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} \)  

Model: \( h_\theta(x) = \theta_0 + \theta_1 x \)
The cost (or loss) function

We try to find parameters $\hat{\theta} \in \mathbb{R}^2$ such that the cost function $J(\theta)$ is minimal:

$$J : \mathbb{R}^2 \rightarrow \mathbb{R}$$

$$\hat{\theta} = \arg \min_{\theta \in \mathbb{R}^2} J(\theta)$$

Mean Square Error:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left( h_{\theta}(x(i)) - y(i) \right)^2 = \frac{1}{2m} \sum_{i=1}^{m} \left( \theta_0 + \theta_1 x(i) - y(i) \right)^2$$

where $m$ is the number of data points in the training set.
The cost (or loss) function

- We try to find parameters $\hat{\theta} \in \mathbb{R}^2$ such that the cost function $J(\theta)$ is minimal:

$$J : \mathbb{R}^2 \to \mathbb{R}$$

$$\hat{\theta} = \arg \min_{\theta \in \mathbb{R}^2} J(\theta)$$

- Mean Square Error:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left( h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

where $m$ is the number of data points in the training set.
The cost (or loss) function

- We try to find parameters \( \hat{\theta} \in \mathbb{R}^2 \) such that the cost function \( J(\theta) \) is minimal:

\[
J : \mathbb{R}^2 \rightarrow \mathbb{R}
\]

\[
\hat{\theta} = \arg \min_{\theta \in \mathbb{R}^2} J(\theta)
\]

- Mean Square Error:

\[
J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left( h_{\theta}(x^{(i)}) - y^{(i)} \right)^2
\]

\[
= \frac{1}{2m} \sum_{i=1}^{m} \left( \theta_0 + \theta_1 x^{(i)} - y^{(i)} \right)^2
\]

where \( m \) is the number of data points in the training set.
The cost (or loss) function

- We try to find parameters $\hat{\theta} \in \mathbb{R}^2$ such that the cost function $J(\theta)$ is minimal:

$$J : \mathbb{R}^2 \rightarrow \mathbb{R}$$

$$\hat{\theta} = \arg \min_{\theta \in \mathbb{R}^2} J(\theta)$$

- Mean Square Error:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left( h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

$$= \frac{1}{2m} \sum_{i=1}^{m} \left( \theta_0 + \theta_1 x^{(i)} - y^{(i)} \right)^2$$

where $m$ is the number of data points in the training set.
The cost (or loss) function

\[ y = -5.00 + 1.50x \]

\[ J\left( \begin{bmatrix} -5.00 \\ 1.50 \end{bmatrix} \right) = 6.1561 \]
The cost (or loss) function

\[ y = -6.00 + 2.00x \]

\[ J(\begin{bmatrix} -6.00 \\ 2.00 \end{bmatrix}) = 19.3401 \]
The cost (or loss) function

\[ y = -2.50 + 1.00x \]

\[ J\left( \begin{bmatrix} -2.50 \\ 1.00 \end{bmatrix} \right) = 4.7692 \]
The cost (or loss) function

\[ y = -3.90 + 1.19x \]

\[ J(\begin{bmatrix} -3.90 \\ 1.19 \end{bmatrix}) = 4.4775 \]
The cost (or loss) function

So, how do we find \( \hat{\theta} = \arg \min_{\theta \in \mathbb{R}^2} J(\theta) \) computationally?
The cost (or loss) function

So, how do we find $\hat{\theta} = \arg \min_{\theta \in \mathbb{R}^2} J(\theta)$ computationally?
(Stochastic) gradient descent

\[
\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j
\]
\( \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \) for each \( j \)

Step 0, \( \alpha = 0.01 \)
(Stochastic) gradient descent

\[ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j \]

Step 1, \( \alpha = 0.01 \)
(Stochastic) gradient descent

\[ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j \]

Step 20, \( \alpha = 0.01 \)
(Stochastic) gradient descent

\[
\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j
\]

Step 200, \( \alpha = 0.01 \)
(Stochastic) gradient descent

\[ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j \]

Step 10000, \( \alpha = 0.01 \)
(Stochastic) gradient descent

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j$$

Step 10000, $\alpha = 0.005$
(Stochastic) gradient descent

\[
\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j
\]

Step 10000, \( \alpha = 0.02 \)
(Stochastic) gradient descent

\[ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta) \text{ for each } j \]

Step 10, \( \alpha = 0.025 \)
Backpropagation

How do we calculate \( \frac{\partial}{\partial \theta_j} J(\theta) \)?

In other words:
how sensitive is the loss function to the change of a parameter \( \theta_j \)?

why backpropagation?
we could do this by hand for linear regression...
but what about complex functions?
→ *propagate error backward*
(special case of *automatic differentiation*)
Backpropagation

Applying the chain rule:

\[ \frac{\partial e}{\partial b} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b} + \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b} = 1 \cdot 2 + 1 \cdot 3 = 5 \]

Next, let's use dynamic programming to avoid re-computing intermediate results...
Backpropagation

Applying the chain rule:

\[ \frac{\partial e}{\partial b} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b} + \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b} = 1 \cdot 2 + 1 \cdot 3 = 5 \]

Next, let's use dynamic programming to avoid re-computing intermediate results...

Christopher Olah

[Image: Diagram of a computation graph with nodes labeled as follows:
- Node a: \(a = 2\)
- Node b: \(b = 1\)
- Node c: \(c = a + b\), \(c = 3\)
- Node d: \(d = b + 1\), \(d = 2\)
- Node e: \(e = c \times d\), \(e = 6\)
Backpropagation

Applying the chain rule:

\[
\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b} + \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b} = 1 \cdot 2 + 1 \cdot 3 = 5
\]

Next, let's use dynamic programming to avoid re-computing intermediate results...
Backpropagation

applying chain rule:

\[
\frac{\partial e}{\partial b} = \frac{\partial e}{\partial c} \cdot \frac{\partial c}{\partial b} + \frac{\partial e}{\partial d} \cdot \frac{\partial d}{\partial b} = 1 \cdot 2 + 1 \cdot 3 = 5
\]

next, let’s use dynamic programming to avoid re-computing intermediate results...
Backpropagation

forward-mode differentiation lets us compute partial derivatives $\frac{\partial e}{\partial b}$ for all nodes $x$
→ still inefficient if you have many inputs
Backpropagation

backward-mode differentiation lets us efficiently compute $\frac{\partial e}{\partial x}$ for all inputs $x$ in one pass
→ also known as error backpropagation

R. Sennrich

MT – 2018 – 02
To summarize what we have learned

When approaching a machine learning problem, we need:

- a suitable model; (here: a linear model)
- a suitable cost (or loss) function; (here: mean square error)
- an optimization algorithm; (here: a variant of SGD)
- the gradient(s) of the cost function (if required by the optimization algorithm).
To summarize what we have learned

When approaching a machine learning problem, we need:

- a suitable model;
To summarize what we have learned

When approaching a machine learning problem, we need:

- a suitable model;
- a suitable cost (or loss) function;
To summarize what we have learned

When approaching a machine learning problem, we need:

- a suitable model;
- a suitable cost (or loss) function;
- an optimization algorithm;
To summarize what we have learned

When approaching a machine learning problem, we need:

- a suitable model;
- a suitable cost (or loss) function;
- an optimization algorithm;
- the gradient(s) of the cost function (if required by the optimization algorithm).
To summarize what we have learned

When approaching a machine learning problem, we need:

- a suitable model; (here: a linear model)
- a suitable cost (or loss) function; (here: mean square error)
- an optimization algorithm; (here: a variant of SGD)
- the gradient(s) of the cost function (if required by the optimization algorithm).
What is a Neural Network?

A complex non-linear function which:

- is built from simpler units (neurons, nodes, gates, ...)
- maps vectors/matrices to vectors/matrices
- is parameterised by vectors/matrices

Why is this useful?

- very expressive
- can represent (e.g.) parameterised probability distributions
- evaluation and parameter estimation can be built up from components
- relationship to linear regression
- more complex architectures with hidden units (neither input nor output)

neural networks typically use non-linear activation functions
What is a Neural Network?

- A complex non-linear function which:
  - is built from simpler units (neurons, nodes, gates, ...)
  - maps vectors/matrices to vectors/matrices
  - is parameterised by vectors/matrices

- Why is this useful?
  - very expressive
  - can represent (e.g.) parameterised probability distributions
  - evaluation and parameter estimation can be built up from components
What is a Neural Network?

- A complex non-linear function which:
  - is built from simpler units (neurons, nodes, gates, …)
  - maps vectors/matrices to vectors/matrices
  - is parameterised by vectors/matrices

- Why is this useful?
  - very expressive
  - can represent (e.g.) parameterised probability distributions
  - evaluation and parameter estimation can be built up from components

relationship to linear regression

- more complex architectures with *hidden* units
  (neither input nor output)
- neural networks typically use non-linear activation functions
An Artificial Neuron

- $\mathbf{x}$ is a vector input, $\mathbf{y}$ is a scalar output
- $\mathbf{w}$ and $b$ are the parameters ($b$ is a bias term)
- $g$ is a (non-linear) activation function
Why Non-linearity?

Functions like XOR cannot be separated by a *linear* function.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(neurons arranged in layers, and fire if input is $\geq 1$)
Functions like XOR cannot be separated by a linear function.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(neurons arranged in layers, and fire if input is $\geq 1$)
Why Non-linearity?

Functions like XOR cannot be separated by a linear function

XOR

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(neurons arranged in layers, and fire if input is $\geq 1$)
Functions like XOR cannot be separated by a *linear* function.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(neurons arranged in layers, and fire if input is $\geq 1$)
Activation functions

- desirable:
  - differentiable (for gradient-based training)
  - monotonic (for better training stability)
  - non-linear (for better expressivity)
we can use linear algebra to formalize our neural network:

the network

\[
\begin{align*}
    w_1 &= \begin{bmatrix} 1 & 0 \\ 0.5 & 0.5 \\ 0 & 1 \end{bmatrix}, \\
    h_1 &= \begin{bmatrix} A \\ B \\ C \end{bmatrix}, \\
    x &= \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \\
    w_2 &= \begin{bmatrix} 1 & -2 & 1 \end{bmatrix}, \\
    y &= \begin{bmatrix} D \end{bmatrix}
\end{align*}
\]

calculation of \( x \mapsto y \)

\[
\begin{align*}
    h_1 &= \varphi(xw_1) \\
    y &= \varphi(h_1w_2)
\end{align*}
\]
import numpy as np

#activation function
def phi(x):
    return np.greater_equal(x, 1).astype(int)

def nn(x, w1, w2):
    h1 = phi(np.dot(x, w1))
    y = phi(np.dot(h1, w2))
    return y

w1 = np.array([[1, 0.5, 0], [0, 0.5, 1]])
w2 = np.array([[1], [-2], [1]])
x = np.array([1, 0])
print nn(x, w1, w2)
More Complex Architectures

Convolutional

Recurrent

[Kalchbrenner et al., 2014]

Andrej Karpathy

http://karpathy.github.io/2015/05/21/rnn-effectiveness/
Practical Considerations

- **efficiency:**
  - GPU acceleration of BLAS operations
  - perform SGD in mini-batches

- **hyperparameters:**
  - number and size of layers
  - minibatch size
  - learning rate
  - ...

- initialisation of weight matrices
- stopping criterion
- regularization (dropout)
- bias units (always-on input)
Toolkits for Neural Networks

What does a Toolkit Provide

- Multi-dimensional matrices (tensors)
- Automatic differentiation
- Efficient GPU routines for tensor operations

Torch  http://torch.ch/
TensorFlow  https://www.tensorflow.org/
Theano  http://deeplearning.net/software/theano/

There are many more!
Further Reading

some slides borrowed from:

- Sennrich, Birch, and Junczys-Dowmunt (2016): Advances in Neural Machine Translation
- Sennrich and Haddow (2017): Practical Neural Machine Translation
A Convolutional Neural Network for Modelling Sentences.